Aligning Alignments Exactly

(Extended abstract)

John Kececioglu* and Dean Starrett†
Department of Computer Science
The University of Arizona
Tucson AZ 85721, USA

ABSTRACT

A basic computational problem that arises in both the construction and local-search phases of the best heuristics for multiple sequence alignment is that of aligning the columns of two multiple alignments. When the scoring function is the sum-of-pairs objective and induced pairwise alignments are evaluated using linear gap-costs, we call this problem *Aligning Alignments*. While seemingly a straightforward extension of two-sequence alignment, we prove it is actually NP-complete. As explained in the paper, this provides the first demonstration that minimizing linear gap-costs, in the context of multiple sequence alignment, is inherently hard.

We also develop an *exact algorithm* for Aligning Alignments that is remarkably efficient in practice, both in time and space. Even though the problem is NP-complete, computational experiments on both biological and simulated data show we can compute optimal alignments for all benchmark instances in two standard datasets, and solve very-large random instances with highly-gapped sequences.

Keywords: Multiple sequence alignment, sum of pairs, linear gap costs, exact algorithms

General Terms: Algorithms

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Computations on discrete structures, pattern matching

1. INTRODUCTION

While it is widely recognized in computational biology that linear gap-costs are necessary to get biologically-correct alignments of *two* sequences [7], in alignment of *multiple* sequences almost without exception the known exact algorithms [37, 5, 22, 23] and approximation algorithms [18, 33, 3, 21, 41, 35] have not considered linear gap-costs. The only exception we know (other than studies of three sequences [8, 13, 34]) is the algorithm behind the software `MSA` [26, 17], which uses linear gap-costs and can potentially compute optimal multiple alignments. Even here, however, the algorithm does not compute *exact* gap-costs: `MSA` uses a heuristic suggested by Altschul [1] that overcounts the true number of gap-initiation events. We believe this inaccuracy in gap-initiation costs is one of the reasons the best heuristics for multiple alignment [45, 6, 4, 38, 17, 2, 36] place gaps poorly on tough instances [29].

Computing correct gap-counts in exact algorithms for multiple alignment seems to unavoidably add exponential-overhead to algorithms that are already exponential in the worst case. (In `MSA`, for example, the design decision to just include heuristic gap-counts increased the time complexity for k sequences by a factor of 2^k in the worst case [17].) Nevertheless, the fact that researchers have been unable to compute exact gap-counts efficiently, or even that multiple alignments can end in an exponential number of gap configurations [1], does not prove it cannot be done. How to determine the inherent complexity of handling exact gap-counts has been a mystery though, since the standard formulations of multiple sequence alignment are already NP-complete *without* linear gap-costs [28, 22, 42, 43].

In this paper, along with several other results, we resolve this dilemma. We show for the first time that there is a form of multiple alignment which we call *Aligning Alignments* that is:

(1) NP-complete *with* gap counts,
(2) polynomial-time solvable *without* them, yet
(3) can be exactly solved with gap counts in practice.

Together (1) and (2) establish that exact gap-counts are inherently hard in the *worst case* (independent of the complexity of multiple alignment) while (3) shows they are tractable in *practice*.

Aligning Alignments is the problem of finding an optimal alignment of the columns of two multiple sequence alignments under the sum-of-pairs objective with linear gap-costs. The *sum-of-pairs* objective [5] scores a multiple alignment by the sum of the scores of the two-sequence alignments induced on all pairs of sequences. With *linear gap-costs* a run of either x insertions or deletions costs $\gamma + \lambda x$, where γ is the gap-initiation cost and λ is the gap-extension cost. For $\gamma = 0$

*Contact author. Research supported by the US National Science Foundation through CAREER Award DBI-0196202 and Grant DBI-0317498. Email: `kece@cs.arizona.edu`

†Research supported by a US Department of Education GAANN Fellowship from Award P200A000302, and a US National Science Foundation IGERT Fellowship from the University of Arizona Program in Genomics. Email: `dms@cs.arizona.edu`

RECOMB'04, March 27–31, 2004, San Diego, California, USA.

(when gap counts are irrelevant) the problem can be solved in polynomial time, since it is then equivalent to aligning two sequences where we treat the columns of the alignments as generalized letters. Our *negative* result is that for $\gamma > 0$ (when gap counts matter) the problem is NP-complete.

Our *positive* result is an exact algorithm for Aligning Alignments that is fast in practice. Suppose the input is a $k \times m$ alignment A versus an $\ell \times n$ alignment B. The algorithm uses dynamic programming over the columns of A and B, where at each entry (i, j) in the table we keep a list of *shapes*. A shape is an ordered partition of the $k+\ell$ sequences that represents the final gap-configuration of a multiple alignment on the columns in prefixes $A[1:i]$ and $B[1:j]$. In the worst case, the number of shapes at an entry is exponential in k and ℓ. We develop two techniques to prune the list of shapes, one based on a cost lower-bound and the other based on the gap configuration, which keep the shape-lists surprisingly small in practice. Computational experiments show we can solve all instances of Aligning Alignments obtained from the benchmark alignments in the datasets of McClure, Vasi and Fitch [29] and Thompson, Plewniak and Poch [40], as well as highly-gapped random instances with 250 sequences and 500 columns in each alignment.

Related work Gotoh, in two pioneering papers [14, 15], first considered an approach of this type, though his description is not easy to follow. By comparison our algorithm is simple, provably correct, and considers fewer shapes as shown in Section 5.3. While Gotoh does not analyze the running time of his procedure, our NP-completeness result shows that unless P = NP, it cannot run in polynomial time.

Kececioglu and Zhang [25] developed algorithms that solve a relaxed version of Aligning Alignments with approximate gap counts (that they termed *optimistic* or *pessimistic*) for several variations of the problem where either of the input alignments are a single sequence, a multiple alignment, or a weighted profile. They also developed an efficient algorithm that solves the special case of Aligning Alignments where one of the inputs is a string and the other is a multiple alignment, while conjecturing that in general Aligning Alignments is NP-complete.

During the preparation of this paper we learned that Ma, Wang and Zhang [27] have also recently and independently resolved the conjecture of Kececioglu and Zhang and proved that Aligning Alignments is NP-complete. Our reduction and proof are significantly simpler, and also yield a stronger result: that the problem remains NP-complete even when all strings have at most five letters, at most one internal gap, and edit operations have unit cost.

Plan of the paper In the next section we prove Aligning Alignments is NP-complete. Section 3 then develops an exact algorithm that computes an optimal alignment. Section 4 analyzes the worst-case time and space for this algorithm. Section 5 presents two techniques for reducing the time, which are remarkably effective in practice. Section 6 shows that one of these techniques, dominance pruning, can be implemented to run in space linear in the number of columns in the input. Section 7 then presents results from extensive experiments on both biological and simulated data that show we can compute optimal alignments for all instances in two standard benchmark datasets, as well as for very-large random instances with highly-gapped sequences.

This paper is an extended abstract of [24].

2. COMPUTATIONAL COMPLEXITY

We now formally define our problem. Let $\mathcal{S}$ be a collection of strings $S_1, \ldots, S_k$ over alphabet Σ. Characters in Σ are called *letters*. Let '-' be a character not in Σ called a *spacer*. An *alignment* of $\mathcal{S}$ is a matrix A with k rows where (i) each entry is either a letter or a spacer, (ii) no column is all spacers, and (iii) reading across row i and removing spacers spells string S_i. We call A a *multiple* alignment when we want to emphasize that $\mathcal{S}$ may have more than two strings.

With *linear gap costs*, we score a two-string alignment as follows. A *gap* is a maximal run of columns of the form $\frac{a}{-}$ or $\frac{-}{a}$ (but not both). The *length* of a gap is its number of columns. The *gap cost* for a gap of length x is $\gamma + \lambda x$, where constant $\gamma \geq 0$ is the *gap initiation* cost and constant $\lambda \geq 0$ is the *gap extension* cost. A *substitution cost* function δ assigns each pair of letters a, b the cost $\delta(a, b) = \delta(b, a)$. The *alignment cost* is the sum of the substitution costs $\delta(a, b)$ over all nongap columns $\frac{a}{b}$, plus the sum of the gap costs. Alternately we can extend the substitution cost function by defining $\delta(a, -) = \delta(-, a) = \lambda$ for all letters a. The alignment cost is then equal to the sum of δ over all columns, plus a term γy where y is the total *gap count* for the alignment. The triple $(\delta, \gamma, \lambda)$ specifies the cost function f for two-string alignments.

Given a cost function f on two-string alignments, the *sum-of-pairs objective* [5] scores a multiple alignment A as follows. Each pair of rows i, j of A induces a two-string alignment A_{ij} of S_i, S_j. (Columns of A_{ij} where both rows have spacers are ignored.) A *weight* function ω assigns each pair of strings S_i, S_j the weight $\omega(i, j) = \omega(j, i)$. The *sum-of-pairs cost* of A is the weighted sum of the costs under f of the two-string alignments induced by all unordered pairs of rows, where the cost of A_{ij} is weighted by $\omega(i, j)$. The pair (ω, f) specifies the sum-of-pairs objective function.

DEFINITION 1 (ALIGNING ALIGNMENTS). *The Aligning Alignments Problem is the following. The input is multiple alignments A and B, weight function ω on pairs of strings from A and B, substitution cost function δ, gap initiation cost γ, and gap extension cost λ. The output is an alignment of the columns of A versus the columns of B that minimizes the sum-of-pairs objective with linear gap costs.*

In other words, the alignments A and B are viewed as two *sequences*. The problem is to align these two sequences by inserting, deleting, and substituting *columns*. A solution yields a multiple alignment C of the strings in A and B, where the subalignment of C induced by the strings in either A or B is fixed. The problem evaluates C using the sum-of-pairs objective, scoring the induced two-string alignments with linear gap costs.

When $\gamma = 0$ (so gap counts have no effect) the problem is solvable in polynomial time. In this case we can reduce the problem to standard alignment of two strings, by treating the columns of A and B as generalized letters and defining generalized insertion, deletion, and substitution costs.

Surprisingly when $\gamma > 0$ (so gap counts matter) the problem is provably hard, as we now show. The *decision version* of Aligning Alignments has an extra input parameter: a cost bound d. The problem is to decide whether A and B have an alignment of cost at most d.

THEOREM 1 (COMPLEXITY). *The decision version of Aligning Alignments is NP-complete.*

PROOF SKETCH. To show the problem is NP-hard, we use a reduction from the Maximum Cut Problem [9]. Recall that an instance of Maximum Cut is an undirected edge-weighted graph $G = (V, E)$ (without self-loops or parallel edges) and an integer c. The problem is to determine whether there is a bipartition of vertices V into two nonempty subsets L and R, called a *cut*, such that the sum of the weights of the edges in E that *cross* the cut (by having one end in L and the other in R) is at least c. Maximum Cut remains NP-complete when all edges have unit weight [10], so we use this simplified form of the problem. In this case c is a lower bound on the number of edges that cross the cut, which we call the *cardinality* of the cut.

Given an instance (V, E, c) of Maximum Cut, we construct an instance $(A, B, \omega, \delta, \gamma, \lambda, d)$ of Aligning Alignments as follows. Let $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$. Alignments A and B are over the binary alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$. They have two types of rows (*edge* rows and *dummy* rows), and two types of columns (*vertex* columns and *dummy* columns). Both alignments consist of m edge rows followed by k dummy rows, where $k = \Theta(m)$ is a parameter we determine later. Columns in A and B are in n consecutive *groups* corresponding to the n vertices. Groups start with a dummy column, followed by one vertex column in A and two vertex columns in B. The two vertex columns in the group corresponding to vertex v in B represent the two sides of a cut where v may fall. Intuitively, dummy columns confine the alignment so a vertex column in A is aligned to one of the two corresponding vertex columns in B; dummy rows ensure dummy columns for corresponding groups are aligned.

The entries in a row are as follows. The row in A for edge $e_i = (v_j, v_{j'})$ has `1`'s in the vertex columns for v_j and $v_{j'}$ and spacers everywhere else. Assuming $j < j'$, the corresponding edge row in B has substring `010` at the group for v_j, substring `-10` at the group for $v_{j'}$, and spacers everywhere else. Dummy rows are $(\mathtt{0-})^n$ in A and $(\mathtt{0--})^n$ in B.

Weight function ω gives all pairs of rows unit weight. The substitution cost function δ has two values: $\delta(\mathtt{0},\mathtt{0}) = \delta(\mathtt{1},\mathtt{1}) = 0$, and $\delta(\mathtt{0},\mathtt{1}) = \delta(\mathtt{1},\mathtt{0}) = \sigma$. Thus the alignment cost function is given by three constants: γ, λ, and σ. Fixing γ, λ, σ, k, and d specifies the constructed instance of Aligning Alignments.

We say an alignment of A and B is in *normal form* if every dummy column of A is aligned to its corresponding dummy column in B, and every vertex column of A is aligned to one of its two corresponding vertex columns in B. We denote the vertex columns of A and B associated with vertex v by A^v, B^{v^L}, and B^{v^R}.

With a cut (L, R) of G we associate a normal-form alignment C of A and B as follows. For each vertex v, align column A^v with column B^{v^L} in C if $v \in L$; otherwise align A^v with B^{v^R}. In the full proof [24] we show that G has a cut of cardinality at least c if and only if A and B have an alignment in normal form of cost at most d, when we fix $\gamma = 2$, $\lambda = 1$, $\sigma = 1$, and $d = \chi - c$ where χ is a constant that depends only on G. We then show that choosing $k = 5m$ ensures every alignment of A and B of cost at most d is in normal form, which completes the proof. □

The proof yields a strong result: Aligning Alignments remains NP-complete for *unweighted sum-of-pairs* (where all pairs of rows have the same weight), for input strings over a *binary alphabet* (hence for DNA and protein sequences), and for the *unit-cost metric* (the simplest cost function, where insertion, deletion, and substitution cost 1 and an identity costs 0).

To understand what makes the problem hard, it is worth asking how much more we can constrain Aligning Alignments and still have an NP-complete problem. Given that both alignments are riddled with gaps in the reduction, it might appear that multiple gaps makes the problem hard. Notice however that by spreading the `0`'s of each dummy row across n rows, we can eliminate interior gaps in dummy rows and break up their length. The proof still goes through, but now every row contains a string of constant length with a constant number of gaps. Thus Aligning Alignments remains NP-complete when the alignments are over strings of length at most 5 and every row has at most 3 gaps, with at most 1 gap in the interior of each string.

Interestingly if *one* of A or B has no gaps, Aligning Alignments is solvable in polynomial time. One way to see this is to reduce the problem to aligning a *string* against an *alignment*, by treating the gapless alignment as a string of columns, which Kececioglu and Zhang [25] solve in $O(km^2 + mn \log m)$ time for a $k \times m$ alignment versus an n-length string, assuming a constant-size alphabet. Our general exact algorithm of the next section also runs in polynomial time in this special case, but with a larger time bound.

3. EXACT ALGORITHM

We now develop an exact algorithm for Aligning Alignments that finds an optimal solution by dynamic programming. Our presentation ignores the weights ω on pairs of sequences, or equivalently assumes uniform weights. All our results apply to nonuniform weights, and incorporating them by scaling the pairwise alignment costs is straightforward.

The algorithm is conceptually simple, though presenting it requires a fair amount of formalism. Essentially the algorithm follows a standard dynamic programming approach by viewing the alignments as sequences of columns where a subproblem, which usually corresponds to a prefix of both sequences, now has an additional parameter that specifies the shape of the gaps in which an optimal alignment ends.

Deriving a recurrence based on shapes To evaluate the cost of an alignment with linear gap-costs we need to determine the number of gaps initiated by a column. Whether a column starts a gap in a pair of rows depends on the relative order of the rightmost letter in each row. We capture this relative order in what we call the shape of an alignment. Formally, a *shape* s for an alignment with k rows is an ordered partition

$$s = (s_1, s_2, \ldots, s_p),$$

where $1 \le p \le k$. The s_i form a *partition* of the row set $\{1, \ldots, k\}$, or in other words they are disjoint subsets whose union is all rows. Furthermore the s_i, which we call the *blocks* of the partition, are *ordered*: s_i precedes s_{i+1}. The interpretation of a shape is that each block represents a set of rows whose rightmost letters occur in the same column, and that these columns occur across the alignment in the same order as the blocks in the shape. So for a pair of rows i and j, the relative order of their last letters is given by the relative order of the blocks that contain i and j.

We say an alignment ends with rows i and j *flush* if the rightmost letter in each row occurs in the same column, or

both rows have no letters. Otherwise the rightmost letter in row i is to the right or left of the rightmost letter in row j, in which case we say i *overhangs* or *underhangs* j.

For example, shape $(\{1,3\}, \{2,4\})$ means the associated alignment has 4 rows. The alignment ends with rows 2 and 4 flush; rows 1 and 3 end earlier and are also flush. In the induced pairwise alignments, row 2 overhangs row 3, and row 1 underhangs row 2.

Suppose the input to our problem is a $k \times m$ multiple alignment A and an $\ell \times n$ multiple alignment B. We denote the entry in A at row i and column j by $A[i,j]$. We write $A[j]$ to denote column j of A, and $A[j:j']$ to denote the subalignment of A consisting of columns j through j' when $j \le j'$, and the empty alignment otherwise.

To find an optimal alignment of A and B, the subproblem we solve is to determine for a given shape s and indices $0 \le i \le m$ and $0 \le j \le n$, the cost of an optimal alignment of the prefixes $A[1:i]$ and $B[1:j]$ that ends in shape s. We call the solution cost for this subproblem $C(i,j,s)$. In general not every shape s can be realized by alignments of given prefixes. We denote the set of shapes of all aligments of these prefixes by $\mathcal{S}(i,j)$. The cost of an optimal alignment of A and B is then

$$\min_{s \in \mathcal{S}(m,n)} \Big\{ C(m,n,s) \Big\}.$$

To count gaps we use the following predicates. For a pair of rows p and q in an alignment with shape s,

- ${}_q s^p$ if and only if p overhangs q in the alignment, and
- ${}^p s_q$ if and only if p underhangs q.

For rows p and q and column c,

- ${}_q c^p$ if and only if p has a letter and q has a spacer in column c, and
- ${}^p c_q$ if and only if q has a letter and p has a spacer.

For alignment columns a and b, let (a,b) be the new column obtained by placing a on top of b. When using this notation, a and b will be either columns from A or B or the column of all spacers, which we denote by `-`. In terms of the above predicates, the total *number of gaps* initiated by appending column (a,b) onto an alignment that ends in shape s is

$$g(a,b,s) \;:=\; \sum_{\substack{p \in A \\ q \in B}} \Big(\big({}_q(a,b)^p \text{ and } \overline{{}_q s^p} \big) \text{ or } \big({}^p(a,b)_q \text{ and } \overline{{}^p s_q} \big) \Big).$$

(In evaluating the sum, `true` maps to 1 and `false` maps to 0, while $\overline{x}$ denotes the logical negation of x.)

We now give a recurrence for $\mathcal{S}(i,j)$, the set of shapes of all alignments of a prefix of the input. For shape s and column c, the shape obtained by concatenating c onto any alignment ending in s is denoted $s \circ c$. For a set S of shapes, $S \circ c$ denotes $\{s \circ c \;:\; s \in S\}$. We denote the *flat shape*, in which all rows are flush (so the associated alignment ends with a column of all letters, or the alignment is empty), by

$$\varphi \;:=\; \Big(\{1, \ldots, k+\ell\} \Big).$$

In this notation, for $i \le m$ and $j \le n$,

$$\mathcal{S}(i,j) \;=\; \begin{cases} \{\}, & i < 0 \text{ or } j < 0; \\ \{\varphi\}, & i = 0 \text{ and } j = 0; \\ \mathcal{S}(i-1,j) \circ (A[i], \text{-}) & \\ \quad \bigcup\, \mathcal{S}(i,j-1) \circ (\text{-}, B[j]) & \\ \quad \bigcup\, \mathcal{S}(i-1,j-1) \circ (A[i], B[j]), & \text{otherwise.} \end{cases}$$

To derive a recurrence for $C(i,j,s)$, the key observation is that an optimal alignment of this prefix of the input that ends in shape s must have some final column c, and removing c must yield an optimal alignment ending in shape $\widetilde{s}$ where $\widetilde{s} \circ c = s$. By this observation, for $0 \le i \le m$, $0 \le j \le n$, $t \in \mathcal{S}(i,j)$, and $(i,j) \ne (0,0)$,

$$C(i,j,t) \;=\; \min \begin{cases} \min\limits_{\substack{s \in \mathcal{S}(i-1,j) \\ s \circ (A[i], \text{-}) = t}} \left\{ \begin{array}{l} C(i-1,j,s) \\ +\, \gamma\, g\big(A[i], \text{-}, s\big) \\ +\, \lambda \ell \,\big|A[i]\big| \end{array} \right\}, \\ \min\limits_{\substack{s \in \mathcal{S}(i,j-1) \\ s \circ (\text{-}, B[j]) = t}} \left\{ \begin{array}{l} C(i,j-1,s) \\ +\, \gamma\, g\big(\text{-}, B[j], s\big) \\ +\, \lambda k \,\big|B[j]\big| \end{array} \right\}, \\ \min\limits_{\substack{s \in \mathcal{S}(i-1,j-1) \\ s \circ (A[i], B[j]) = t}} \left\{ \begin{array}{l} C(i-1,j-1,s) \\ +\, \gamma\, g\big(A[i], B[j], s\big) \\ +\, \sum\limits_{\substack{p \in A \\ q \in B}} \delta\big(A[p,i], B[q,j]\big) \end{array} \right\}, \end{cases}$$

where $|c|$ denotes the number of letters in column c. For $(i,j) = (0,0)$,

$$C(0,0,\varphi) \;:=\; 0.$$

Solving the corresponding shortest-paths problem As is standard, we view the process of evaluating the recurrence as an equivalent shortest-paths problem in a grid-structured graph G. Graph G has a vertex for every subproblem (i,j,s) where s is in $\mathcal{S}(i,j)$, and an edge directed from vertex $(\widetilde{\imath}, \widetilde{\jmath}, s)$ to (i,j,t) if $C(\widetilde{\imath}, \widetilde{\jmath}, s)$ appears in the above recurrence for $C(i,j,t)$. Every edge then corresponds to a column c where $s \circ c = t$, and is weighted by the alignment cost of column c including gap initiation costs. In general a vertex (i,j,s) has three out-edges, corresponding to appending an insertion, deletion, or substitution column. Finding a shortest path in G from source vertex $(0,0,\varphi)$ to the sink vertices (m,n,t) for t in $\mathcal{S}(m,n)$ finds an optimal alignment of A and B.

It is advantageous when computing shortest paths to process all the vertices at coordinate (i,j) as a group, as shown in Section 5. Since G is acyclic, we can find shortest paths by considering vertices in any topological order. We examine groups in lexicographic order, which corresponds to evaluating the recurrence in row-major order on entries (i,j). As we encounter entries starting from the source $(0,0,\varphi)$, we effectively construct G on the fly, building at each entry (i,j) a list of all shapes s in $\mathcal{S}(i,j)$, which we call the entry's *shape list* $L(i,j)$. With each shape s on list $L(i,j)$, we maintain the cost of the shortest path known to vertex (i,j,s).

To process an entry (i,j), the algorithm examines each shape on $L(i,j)$. The entire shape list has already been constructed by processing lexicographically earlier entries,

and the cost of a shortest path from the source to each s on $L(i,j)$ is known. The algorithm then considers the three out-edges of vertex $v=(i,j,s)$ to vertices w in the adjacent entries $(i,j{+}1)$, $(i{+}1,j)$, and $(i{+}1,j{+}1)$. If the shape t for w is not already on the shape list for the adjacent entry, it is created, and its associated cost is initialized to the length of the shortest path to w through v. If t is present in the shape list, its cost is updated.

To process entry (m,n), its list $L(m,n)$ is scanned for the shape t with minimum associated cost. On determining t, the shortest path from the source to t is recovered, yielding an optimal alignment of A and B. Interestingly, recovering the shortest path does not require back pointers, since given a shape t at entry (i,j), its last block uniquely identifies the final column of its associated alignment, which in turn specifies the preceding entry $(\widetilde{\imath},\widetilde{\jmath})$ on the path. Once this entry is known, the shape s such that $(\widetilde{\imath},\widetilde{\jmath},s)$ is the preceding vertex on the shortest path can be determined by examining all s on the entry's list and seeing which of these shapes yields shape t and value $C(i,j,t)$ in the recurrence.

This algorithm is equivalent to the standard breadth-first search for single-source shortest paths in acyclic graphs, which takes time linear in the size of the graph. Our graphs have size linear in the number of vertices, which is just the total number of shapes at all entries. The total number of shapes, however, is highly dependent on the gap structure of the inputs A and B, and is not easy to determine. We next determine this quantity in the worst case.

4. TIME AND SPACE

The time and space for the exact algorithm depends on the number of shapes at entries, which we now analyze. (This section is technical and omits some proofs from [24].)

4.1 Number of shapes

We now determine the exact worst-case number of shapes at each entry, which in essence yields the worst-case time and space for the exact algorithm. The result, which is given in Theorem 2, is surprising in that we are able to determine the worst case *precisely* (not just its order of growth) and that it is an *unexpected* function of the input size (not simply the number of ordered partitions of the input sequences, as in pure sum-of-pairs multiple alignment [1]).

Preliminaries We denote the *worst-case number of shapes* at an entry by $S_{ab}(i,j)$. Formally when $a>0$ and $b>0$, define $S_{ab}(i,j)$ to be the maximum number of shapes at entry (i,j) over all inputs A and B, where A has a sequences and at least i columns and B has b sequences and at least j columns. When $a=0$ or $b=0$, define $S_{ab}(i,j)=1$.

We denote the worst-case number of shapes *contributed* to a table entry from its adjacent entries by the following three quantities:

- $V_{ab}(i,j)$, the maximum number of shapes at entry (i,j) whose alignments extend shapes from $(i-1,j)$, or in other words follow a *vertical* edge from $(i-1,j)$ to (i,j), where the maximum is over all inputs with a and b sequences,
- $H_{ab}(i,j)$, the maximum number of shapes at (i,j) that follow a *horizontal* edge from $(i,j-1)$, and
- $D_{ab}(i,j)$, the maximum number of shapes at (i,j) that follow a *diagonal* edge from $(i-1,j-1)$.

The following lemma bounds the number of shapes contributed from a given direction in terms of the number of shapes at the contributing entry with fewer *sequences*. This reduction in the number of sequences is key for the inductive step in the proof of our main result.

LEMMA 1 (CONTRIBUTED SHAPES). *For all $i\ge 0$, $j\ge 0$, $a\ge 1$, and $b\ge 1$,*

$$V_{ab}(i{+}1,j) \;\le\; S_{a-1,b}(i,j), \tag{1}$$

$$H_{ab}(i,j{+}1) \;\le\; S_{a,b-1}(i,j), \tag{2}$$

$$D_{ab}(i{+}1,j{+}1) \;\le\; S_{a-1,b-1}(i,j). \tag{3}$$

We denote the *number of alignments* of two strings of lengths m and n by $F(m,n)$. Equivalently, $F(m,n)$ is the number of paths in the standard edit grid-graph from vertex $(0,0)$ to vertex (m,n). Function F satisfies the recurrence,

$$F(m,n) \;=\; \begin{cases} 1, & m=0 \text{ or } n=0; \\ F(m{-}1,n) & \\ \quad + F(m,n{-}1) & \\ \quad + F(m{-}1,n{-}1), & m>0 \text{ and } n>0. \end{cases}$$

Note that F is monotonic increasing in its arguments.

Results We now have the tools to prove the main result of this section: the number of shapes at an entry is at most $F(a,b)$, the number of alignments of two strings whose lengths are the number of *sequences* in the input. We actually prove the following tighter result, which as it turns out is exact.

LEMMA 2 (UPPER BOUND). *For all $i\ge 0$, $j\ge 0$, $a\ge 1$, and $b\ge 1$,*

$$S_{ab}(i,j) \;\le\; F\big(\min\{a,i\},\, \min\{b,j\}\big).$$

PROOF. When i or j are zero the lemma holds, since $S_{ab}(i,0)=S_{ab}(0,j)=F(m,0)=F(0,n)=1$. So assume nonzero i and j. We use induction on $a+b$.

In the basis, $a+b=2$, which implies A and B are strings. Only three shapes are possible: either A overhangs, underhangs, or is flush with B. Thus $S_{11}(i,j)\le 3 = F(1,1)$, and the lemma holds.

In general for $a+b>2$, assume the lemma holds for all $S_{\tilde{a}\tilde{b}}(i,j)$ where $\tilde{a}+\tilde{b} < a+b$. Let us write $x \bigtriangledown y$ for $\min\{x,y\}$. Since the set of shapes counted by $S_{ab}(i,j)$ is the union of the contributions from the entries adjacent in the vertical, horizontal, and diagonal directions,

$$\begin{aligned} S_{ab}(i,j) &\le V_{ab}(i,j) \;+\; H_{ab}(i,j) \;+\; D_{ab}(i,j) \\ &\le S_{a-1,b}(i{-}1,j) \;+\; S_{a,b-1}(i,j{-}1) \\ &\qquad +\; S_{a-1,b-1}(i{-}1,j{-}1) \\ &\le F\big((a\bigtriangledown i){-}1,\, b\bigtriangledown j\big) \;+\; F\big(a\bigtriangledown i,\, (b\bigtriangledown j){-}1\big) \\ &\qquad +\; F\big((a\bigtriangledown i){-}1,\, (b\bigtriangledown j){-}1\big) \\ &= F\big(\min\{a,i\},\, \min\{b,j\}\big), \end{aligned}$$

where the second inequality follows from Lemma 1, the third from the induction hypothesis, and the fourth from the recurrence for F. □

The bound of Lemma 2 is tight, as the next result shows.

LEMMA 3 (LOWER BOUND). *For all $i \geq 0$, $j \geq 0$, $a \geq 1$, and $b \geq 1$,*

$$S_{ab}(i,j) \;\; \geq \;\; F\big(\min\{a,i\},\, \min\{b,j\}\big).$$

Moreover for every $m \geq 1$ and $n \geq 1$, there is an $a \times m$ and $b \times n$ input that meets this lower bound at every entry of its table.

Combining Lemmas 2 and 3 yields the exact worst-case number of shapes.

THEOREM 2 (SHAPES AT AN ENTRY). *The worst-case number of shapes at entry (i,j) of the dynamic programming table, for inputs with a and b sequences, is exactly*

$$F\big(\min\{a,i\},\, \min\{b,j\}\big),$$

where $F(m,n)$ is the number of alignments of two strings of lengths m and n. Furthermore there is a fixed input that, at every entry of the table, attains the worst case.

The last part of Theorem 2 implies that $\sum_{i,j} S_{ab}(i,j)$ is the worst-case total number of shapes. This governs the worst-case time and space, and has the following order of growth.

COROLLARY 1 (TOTAL SHAPES). *The worst-case total number of shapes in the table, for the exact algorithm on an $a \times m$ and $b \times n$ alignment, is*

$$\Theta\Big(F\big(\widetilde{a}{+}1, \widetilde{b}{+}1\big) + \widetilde{m}\, F\big(\widetilde{a}{+}1, \widetilde{b}\big) + \widetilde{n}\, F\big(\widetilde{a}, \widetilde{b}{+}1\big) + \widetilde{m}\widetilde{n}\, F\big(\widetilde{a}, \widetilde{b}\big)\Big),$$

where $\widetilde{a} = \min\{a,m\}$, $\widetilde{b} = \min\{b,n\}$, $\widetilde{m} = \max\{m{-}a, 0\}$, and $\widetilde{n} = \max\{n{-}b, 0\}$.

With the number of shapes in hand, we can now determine the worst-case time and space for the exact algorithm.

4.2 Worst-case performance

Recall that the exact algorithm proceeds lexicographically through the entries of the dynamic programming table, and at each entry maintains a list of the shapes that can be achieved by alignments that end at that entry. With each shape on the list, the cost of the optimal alignment that ends in that shape is stored. To process entry (i,j), the algorithm considers each shape on its list and extends the alignment to entries $(i,j{+}1)$, $(i{+}1,j)$ and $(i{+}1,j{+}1)$ by inserting, deleting, or substituting a column. Extending to an alignment at one of these adjacent entries yields an alignment ending in some shape s with some cost C. The algorithm searches the list L at the adjacent entry to see whether s is already in L. If s is not present, it is inserted with cost C. If s is present and its cost in L is greater than C, its associated cost is lowered to C.

Suppose we represent a shape s as a one-dimensional array $S[1{:}a{+}b]$ for an input with a and b sequences, where $S[i]$ is the rank of the block in s that contains row i. We can then maintain each shape list as a balanced search tree, where shapes are ordered lexicographically by their array representation. Comparing two shapes by lexicographic order takes $O(a+b)$ time. So searching for a shape, inserting a shape, or updating a shape's cost in a list of length ℓ takes $O((a+b)\log \ell)$ time. Finally when extending an alignment, determining the resulting shape and counting gap initiation events to evaluate its cost takes $O(ab)$ time. In short, if every list has length at most ℓ, processing an entry takes $O(\ell(a+b)\log \ell + \ell ab)$ time and uses $O(\ell(a+b))$ space.

Theorem 2 implies every list has at most $F(a,b)$ shapes. In general, function $F(m,n)$ exhibits a variety of orders of growth depending on the relative order of growth of m and n. When $m = n$, it is known (see Waterman [44, p. 187]) that

$$F(n,n) \;\; = \;\; \Theta\Big(\big(3{+}\sqrt{2}\big)^n \; n^{-1/2}\Big).$$

Combining this fact with the above analysis and Corollary 1 yields the following.

THEOREM 3 (TIME AND SPACE). *To align two alignments, each having k sequences and n columns, the exact algorithm takes worst-case time*

$$\begin{cases} \Theta\Big(\big(3{+}\sqrt{2}\big)^k \; (n{-}k)^2 \; k^{3/2}\Big), & k < n; \\ \Theta\Big(\big(3{+}\sqrt{2}\big)^n \; k^2 \; n^{-1/2}\Big), & k \geq n. \end{cases}$$

This is exactly a factor k greater than the worst-case space.

We can simplify the result of Theorem 3 to the following slightly looser bound. For any constant $c > 3+\sqrt{2} \approx 4.4$, the time for the exact algorithm is

$$O\big(c^{\min(k,n)} \; \max(k,n)^2\big).$$

This transitions between two orders of growth, depending on the relative number of rows and columns in the alignments.

When $m = O(1)$, it is not hard to show $F(m,n) = n^{O(1)}$. By Theorem 2, this implies the following.

COROLLARY 2 (SPECIAL CASE). *When one of the alignments has a constant number of sequences, the exact algorithm runs in polynomial time and space.*

In practice an algorithm whose time and space is exponential in the number of sequences is unlikely to be feasible for large inputs. The next section presents two techniques for reducing the size of a shape list, and hence reducing the running time, which we call *pruning*. While our problem is NP-complete, pruning is remarkably effective in practice, enabling us to compute optimal solutions even for very-large, highly-gapped alignments.

5. REDUCING THE TIME

We now present two techniques for removing shapes from the shape lists of the exact algorithm, while guaranteeing that the algorithm is still correct. The first technique, which we call *dominance pruning*, uses a dominance relation on pairs of shapes. The second, which we call *bound pruning*, exploits upper and lower bounds on the cost of an optimal alignment.

5.1 Dominance pruning

Consider a series of insertions, deletions, and substitutions of columns that extend the alignment at an entry. We call the subalignment given by the series an *extension*. The idea behind dominance pruning is to remove shape t from a list if it can be determined that t is never better than another shape s in the list over all possible extensions. We now derive a sufficient condition for this to hold for a pair of shapes s and t.

For shape s and extension ρ, let $s \circ \rho$ denote the alignment obtained by concatenating ρ onto the alignment associated with s. Let $C(s)$ denote the cost associated with shape s, and $C(\rho)$ the cost of subalignment ρ starting from the flat shape. Finally, let $G(s, \rho)$ count the number of pairs of rows $p \in A$ and $q \in B$ such that p overhangs or underhangs q in a gap that is continued by ρ. Then

$$C(s \circ \rho) \;=\; C(s) \;+\; C(\rho) \;-\; \gamma\, G(s, \rho).$$

So shape t is no better than shape s on extension ρ if

$$\begin{aligned} & C(t \circ \rho) \;-\; C(s \circ \rho) \\ &\quad = \;\; \Big(C(t) - C(s)\Big) \;-\; \gamma\Big(G(t, \rho) - G(s, \rho)\Big) \\ &\quad \geq \;\; \Big(C(t) - C(s)\Big) \qquad (4) \\ &\qquad\quad -\gamma \sum_{\substack{p \in A \\ q \in B}} \Big(\big({}_qt^p \text{ and } \overline{{}_qs^p}\big) \text{ or } \big({}^pt_q \text{ and } \overline{{}^ps_q}\big) \Big) \\ &\quad \geq \;\; 0, \end{aligned}$$

where Inequality (4) follows by upper bounding the pairs of rows that contribute positive terms to $G(t, \rho) - G(s, \rho)$ by the number of pairs p and q that satisfy $({}_qt^p$ and $\overline{{}_qs^p})$ or $({}^pt_q$ and $\overline{{}^ps_q})$.

Since Inequality (4) is independent of ρ, rearranging yields the following sufficient condition for $C(t \circ \rho) - C(s \circ \rho) \geq 0$ to hold for *all* extensions ρ.

Definition 2 (Shape Dominance). *Shape s dominates shape t if*

$$C(t) \;\geq\; C(s) + \gamma \sum_{\substack{p \in A \\ q \in B}} \Big(\big({}_qt^p \text{ and } \overline{{}_qs^p}\big) \text{ or } \big({}^pt_q \text{ and } \overline{{}^ps_q}\big) \Big). \quad (5)$$

With *dominance pruning* the exact algorithm proceeds as follows. Whenever a new shape s is considered for propagation to the list L at an entry, it is compared to each shape t in L to determine if s dominates t or t dominates s. (We actually organize L so comparisons can potentially be skipped.) If t is dominated it is removed from L, while if s is dominated it is not added to L.

In general we say s is *as good as t on all extensions* if for every extension ρ,

$$C(s \circ \rho) \;\leq\; C(t \circ \rho).$$

The following lemma is immediate from the discussion preceding Inequality (5).

Lemma 4 (Extending Dominated Shapes). *Suppose shape s dominates shape t. Then s is as good as t on all extensions.*

A consequence of the following lemma is that interleaving pruning with the propagation of shapes to an entry, as is done by the exact algorithm with dominance pruning, is equivalent to pruning only after all shapes are propagated to the entry.

Lemma 5 (Transitivity). *For all shapes r, s, and t, if r dominates s, and s dominates t, then r dominates t.*

The list of shapes at an entry is formed by propagating shapes from three adjacent entries in the table while applying pruning. We say the shape list at an entry is *determined* once propagation from all three entries is finished.

Lemma 6 (Dominance-Pruning Shape-Lists). *Let L be the set of shapes determined by dominance pruning at entry (i, j), and S be the set of shapes of all alignments of $A[1\!:\!i]$ and $B[1\!:\!j]$. For every shape t in $S - L$ there is a shape s in L that is as good as t on every extension.*

We can now prove dominance pruning is correct.

Theorem 4 (Correctness of Dominance Pruning). *The exact algorithm with dominance pruning finds an optimal alignment.*

Proof. We use induction on the number of columns in an optimal alignment. Once the exact algorithm with dominance pruning determines the list L at entry (m, n), by Lemma 6 applied to the empty extension there is a shape t in L that corresponds to an optimal alignment. Recall from Section 3 that to recover a solution the exact algorithm identifies the last column of the alignment solely from t and the indices of its table entry. Thus the algorithm will find the last column c of an optimal alignment.

Shapes are never removed from a determined list, and a list is determined before it propagates any shapes. So the shape s that propagated t is still in the table and will be identified by the recovery phase. Shape s corresponds to the prefix of the optimal alignment with one less column. By induction this prefix will be recovered, and concatenating c recovers an optimal alignment. □

As demonstrated in Section 7, the exact algorithm with dominance pruning is highly effective. Section 6 shows it also has the desirable property that it can be implemented to run in space *linear* in the number of input columns.

5.2 Bound pruning

With *bound pruning*, a shape is pruned similar to the way subproblems are pruned in branch-and-bound algorithms. We compute a lower bound $L(s)$ on the cost of the best alignment of A and B that extends the alignment associated with shape s. This is compared to an upper bound U on the cost of an optimal alignment of A and B. If $L(s) \geq U$, shape s can be safely pruned.

To obtain upper bound U, at the start of the computation three feasible alignments of A and B are scored, and U is assigned the minimum of these three scores. The three alignments are the so-called *optimistic* and *pessimistic* alignments [25], and the *trivial* alignment of A and B, which aligns column i of A with column i of B possibly with a final run of either insertions or deletions. (The trivial alignment can be good when the input arises from polishing a multiple alignment by splitting it and realigning.)

To compute the lower bound $L(s)$ at an entry, we lookup a score in the dynamic programming table obtained from running the optimistic algorithm on the reverse of the input alignments. The score at each entry in this table is a lower bound on the optimal alignment of a suffix of A with a suffix of B, where these suffixes start with the flat shape. We actually use the three tables computed by the optimistic algorithm, which give the optimistic scores for alignments of the suffixes that start with an insertion, deletion, or substitution. By combining the score of shape s with an optimistic score for the extension of s to an alignment of A and B, and adjusting for gaps that may be overcounted in both scores, a lower bound on the cost of an optimal alignment extending

shape s is obtained. This lower bound is computed separately for extensions starting with an insertion, deletion, or substitution; each must exceed U for s to be pruned.

While dominance pruning always leaves at least one shape in a list, bound pruning can delete an entire list. As shown in Section 7, bound pruning is very effective: often only a tiny fraction of the entries remain with nonempty shape lists. When combined with dominance pruning, it yields our fastest exact algorithm in practice.

5.3 Gotoh's pruning

Gotoh [14] describes a procedure for aligning alignments that uses a form of pruning different from both dominance and bound pruning. (Of our two techniques, Gotoh's is closer to dominance pruning.) We call Gotoh's procedure *extremal pruning*. The precise statement of extremal pruning is long and involves four separate criteria; to save space we omit it. We state here our main result on the relationship between extremal pruning and dominance pruning. (Full details and proofs are in [24].)

Dominance pruning is simpler than extremal pruning, and is also stronger in the following sense. For pruning method A and set of shapes S, let $A(S)$ be the subset of S that remains after pruning by A. We say method A is *stronger* than method B if for all sets S, $A(S) \subseteq B(S)$, and there exist sets S for which $A(S)$ is strictly smaller than $B(S)$.

THEOREM 5 (PRUNING STRENGTH). *Dominance pruning is stronger than extremal pruning.*

6. REDUCING THE SPACE

A classic result in sequence comparison is that an optimal alignment of two strings can be computed in space linear in the lengths of the strings without exceeding the time complexity of the standard quadratic-space algorithm, as discovered by Hirschberg [20]. This result can be generalized to aligning two strings with linear gap-costs, as explained by Myers and Miller [31], though it becomes more complex. Linear-space algorithms are a necessity to optimally align long genomic regions.

Given that the space used by the exact algorithm is potentially exponential (see Theorem 3 in Section 4), reducing its space complexity is critical. In this section we show that Hirschberg's divide-and-conquer approach can be generalized to Aligning Alignments. In particular, the good time-behavior of the exact algorithm with dominance pruning described in Section 5.1 can be retained while achieving space-complexity that grows linearly in the number of columns in the input alignments.

Working out a generalization to Aligning Alignments and showing it is correct in the presence of dominance pruning is involved, and is not possible to present within the allowed page limit. (Full details are in [24].) The key idea is to reduce the subproblem that arises in the divide-and-conquer approach to computing an alignment of minimum cost over an interval $A[i:j]$ and $B[i':j']$ of the input, subject to the constraint that the alignment is preceded by a given shape s and followed by a given shape t. This subproblem can be solved by running the exact algorithm over the interval in both a forward and reverse pass, where each pass maintains shape lists for just two consecutive rows of the table, and then carefully gluing together the terminal shapes of the alignments from both passes. Correctness in the presence of dominance pruning relies on Lemma 6. We state here just the final result.

THEOREM 6 (LINEAR SPACE). *The exact algorithm with dominance pruning can be implemented to find an optimal alignment using space linear in the number of columns in the input, without increasing the time complexity.*

Note that obtaining such a result for bound pruning is obstructed by the fact that it uses quadratic-size tables to lookup lower bounds (though see Myers and Jain [30] for a technique that yields a space-time tradeoff).

7. COMPUTATIONAL RESULTS

7.1 Feasibility of the exact algorithm

In practice we can solve large instances to optimality despite the potential for the number of shapes to grow exponentially in the number of rows. We attribute this to a *ceiling phenomenon* in the growth of shapes. As shown in Section 7.1.2, for instances with a given number of columns the observed number of shapes does not grow once the number of rows exceeds a threshold. Furthermore this threshold is small (roughly 10 rows in our experiments) and appears to be independent of the number of columns.

7.1.1 Time and space

Biological data

Trials were run on challenging biological instances taken from the McClure, Vasi, and Fitch [29] benchmark set, which we call `MVF`, and the Thompson, Plewniak, and Poch [40] set called `BAliBASE`. Trials used all `MVF` benchmarks, and all `BAliBASE` benchmarks in their reference sets 1 through 5, which we call *groups*. These benchmarks are alignments with up to nearly 30 rows and 1,000 columns. Instances were generated by splitting the rows of a benchmark into two subsets. For each `MVF` benchmark, trials were run on all of the roughly 2,000 possible splits. For each `BAliBASE` benchmark, trials were run on 20 random equal-sized splits.

To preview our results, the most time-efficient version of the exact algorithm took less than 2 seconds on each trial, and the most space-efficient version less than 1 megabyte.

Simulated data

A natural question is how big an instance we can solve in a given amount of time and space. Since extrapolating from isolated biological instances is dubious, we ran experiments on simulated data to observe the order of growth as a function of input size.

Generator The data generator was designed with the goal of producing instances at least as hard as typical biological data. Generator parameters include size, a specification of gap structure, and the letter distribution.

We define *spacer density* as the percentage of alignment entries that are spacers. *Startup density* is the percentage of entries that are spacers but not preceded by a spacer. For a given alignment size, startup density specifies the number of gaps; spacer density specifies their lengths. Together spacer- and startup-density are called *gap densities*. The generator produces an alignment with the given gap densities by randomly placing an equal number of gaps in each row, where gaps have the same length to within one unit.

Table 1: Time and space for the BAliBASE benchmarks. For each alignment group and algorithm version, the statistics are for 20 random balanced splits of each alignment in the group, where a benchmark of k sequences was split into alignments with $\lfloor k/2 \rfloor$ and $\lceil k/2 \rceil$ sequences. Versions are: BDQ, bound and dominance pruning in quadratic space, and DL, dominance pruning in linear space.

		Time (sec)									Space (Mb)								
		short			medium			long			short			medium			long		
group	version	mean	max	dev	mean	max	dev	mean	max	dev	mean	max	dev	mean	max	dev	mean	max	dev
1	BDQ	0.00	0.01	0.00	0.02	0.03	0.01	0.05	**0.07**	0.01	0.30	0.35	0.05	1.92	2.48	0.36	5.39	6.74	0.92
	DL	0.03	0.05	0.01	0.25	0.41	0.07	0.70	0.87	0.09	0.01	0.02	0.00	0.03	0.03	0.00	0.03	**0.03**	0.00
2	BDQ	0.01	0.02	0.01	0.08	0.12	0.03	0.19	**0.30**	0.06	0.35	0.39	0.03	2.57	3.02	0.50	6.27	8.45	1.47
	DL	0.13	0.33	0.06	1.30	2.58	0.37	3.36	6.20	1.23	0.02	0.03	0.00	0.07	0.08	0.01	0.11	**0.14**	0.02
3	BDQ	0.01	0.02	0.01	0.11	0.14	0.01	0.23	**0.44**	0.08	0.33	0.39	0.07	2.60	3.17	0.50	6.18	8.35	1.51
	DL	0.18	0.29	0.06	1.85	2.21	0.22	3.48	9.84	1.31	0.03	0.03	0.00	0.08	0.09	0.00	0.11	**0.14**	0.02
4	BDQ							0.33	**1.00**	0.39							10.31	23.17	7.87
	DL							3.08	14.58	4.37							0.07	**0.20**	0.06
5	BDQ				0.11	**0.29**	0.09							3.73	6.72	1.71			
	DL				1.52	3.84	0.77							0.06	**0.12**	0.03			

Table 2: Shape-list lengths for the MVF benchmarks. The columns below give the *maximum* length of a shape list in the table (measured once all shapes have been propagated to an entry), the *total* length of all such shape lists, and the number of shapes *allocated* during the computation. (Fewer shapes are allocated than the total length of all shape lists as shapes are reused during the computation.) Statistics are over all $\Theta(2^k)$ splits of a given alignment of k sequences for a given algorithm version, except for version Q on benchmark kin12, where statistics are for 20 random balanced splits into two alignments of 6 sequences each. A benchmark of size $k \times n$ has k sequences and n columns.

			Maximum length			Total length			Shapes allocated		
benchmark	size	version	mean	max	dev	mean	max	dev	mean	max	dev
glob12	12 × 166	Q	101	129	37	114,088	125,721	7,260	16,996	25,180	3,375
		BQ	9	53	4	421	4,809	227	221	714	35
		DQ	4	6	0	35,157	40,131	1,023	12,067	13,406	760
		BDQ	2	5	0	261	**2,479**	90	184	396	9
		DL	3	5	1	35,690	40,906	1,081	466	**577**	28
pro12	12 × 191	Q	1,836	7,183	1,503	1,043,574	3,177,917	552,556	179,104	794,774	145,675
		BQ	380	4,295	334	11,703	132,500	9,106	2,664	28,421	2,285
		DQ	6	12	1	52,164	57,913	3,469	11,023	15,676	2,105
		BDQ	4	8	1	1,725	**19,113**	780	363	1,783	93
		DL	6	12	1	53,268	58,774	3,578	600	**751**	46
rh12	12 × 194	Q	3,890	8,989	2,672	1,492,059	3,245,298	528,942	161,583	555,116	92,311
		BQ	1,576	8,526	1,050	99,935	282,208	41,871	16,338	59,051	9,193
		DQ	9	20	2	60,018	65,616	2,932	8,922	12,295	922
		BDQ	6	20	2	3,056	**21,078**	1,000	347	2,077	112
		DL	9	20	2	58,899	64,301	2,592	665	**795**	46
kin12	12 × 396	Q	2,635	3,653	1,069	5,790,926	11,950,393	3,015,472	987,011	2,161,794	743,806
		BQ	773	3,653	502	388,099	3,315,026	448,456	102,413	1,065,764	125,554
		DQ	7	16	1	182,149	198,920	7,625	41,365	52,312	4,284
		BDQ	5	16	1	5,277	**39,160**	2,062	905	3,431	241
		DL	7	16	1	193,212	208,494	7,228	1,069	**1,259**	54

The sequences in each row are formed by choosing letters randomly and independently from the given distribution. To test whether random sequences yield easier instances, trials were run on each BAliBASE benchmark in group 1, by randomly shuffling the letters in each row while not moving gaps. The average time and space increased or remained the same compared to unshuffled alignments.

Columns of all spacers were kept in generated data.

Experiments The experiments on simulated data used random instances generated as described above. Both alignments in an instance had the same size, which varied from 1 to 500 rows, and 100 to 10,000 columns. For the letter distribution we took the frequencies observed in BAliBASE. For substitution costs we used the PAM250 matrix [11] with integer entries in the range 0 to 24. Gap initiation and extension costs were fixed at 8 and 12 respectively. The same gap densities were used to generate all instances, so the only variation between instances of a given size is gap placement.

In the benchmarks, startup density ranges from around 1% to 10%, and spacer density from about 5% to 80%, though all benchmarks from MVF and BAliBASE groups 1, 2, and 3 have less than 35% spacer density. To set the gap densities for our experiments, three suites of exploratory tests were run on random alignments with 15 rows and 500 columns generated with startup densities of 1%, 5%, and 10%; for each startup density, spacer density stepped from 5% to 75%.

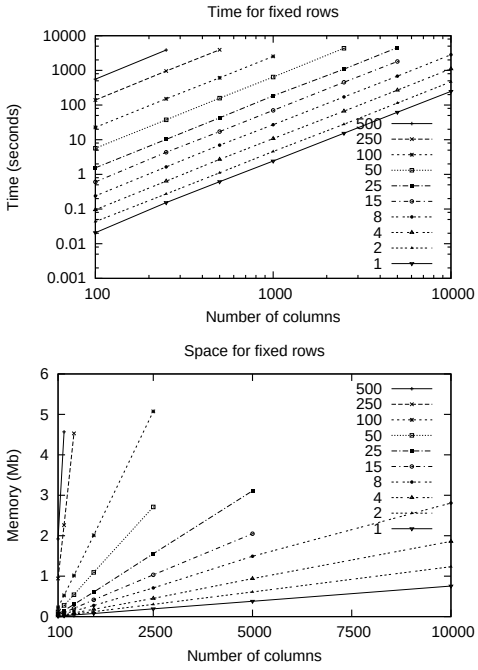

Figure 1: Time and space versus input length for simulated data. Each curve has a fixed number of rows, given in the legend. Each point is an average of 10 trials of version DL. Alignments have 35% spacer density and 10% startup density.

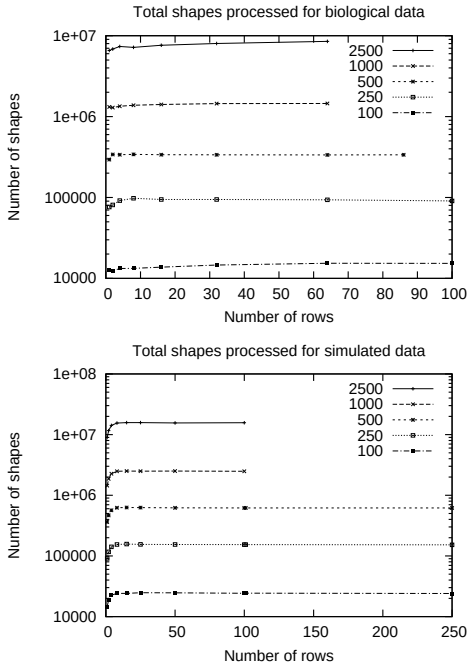

Figure 2: Number of shapes as a function of number of rows for biological and simulated data. In the top plot, curves correspond to BAliBASE benchmarks sh3 (2500, 500), 7tm (1000), and myb (250, 100), trimmed to the number of columns in the legend; instances were generated by randomly selecting rows from the benchmark, without removing null columns. In the bottom plot, curves correspond to simulated alignments with the number of columns in the legend. Trials and gap densities are the same as Figure 1. Total shapes processed is the sum of the lengths of all determined shape lists in the top-level of recursion for version DL.

We chose for startup density the largest value of 10%, since time and space generally increased with startup density. We chose a spacer density of 35%, which covers all benchmarks except those with unusually long inserts, since resource use also increased with spacer density; choosing a higher value would have reduced the size of an instance we could complete within a resource limit, without necessarily showing anything new.

Results

The following *versions* of the exact algorithm were implemented:

- `Q`, no pruning in quadratic space,
- `BQ`, bound pruning in quadratic space,
- `DQ`, dominance pruning in quadratic space,
- `BDQ`, bound and dominance pruning in quadratic space,
- `DL`, dominance pruning in linear space.

For biological data, trials were run on instances derived from the `MVF` and `BAliBASE` benchmarks by splitting their rows into two disjoint subsets, and removing any resulting null columns. For the `BAliBASE` benchmarks, random *balanced* splits of k rows into subsets of size $\lfloor k/2 \rfloor$ and $\lceil k/2 \rceil$ were used. For the `MVF` benchmarks, every possible split was used, except only 20 random balanced splits were used for version `Q` on the `kin12` benchmark.

Time and space results on these instances are summarized for each `BAliBASE` group in Table 1. The boldface entries highlight the maximum *time* used by `BDQ` (our most time-efficient version), and the maximum *space* used by `DL` (our most space-efficient version). Notice that the time for version `DL` is roughly twice that of `DQ`, which is consistent with analysis of their running times.

Table 2 summarizes shape-counts for the `MVF` trials, including the *maximum* length of a shape list in the table, the *total* length of all such lists, and the maximum number of shapes *allocated* at any point in the computation. The total length is a machine-independent measure that captures some aspects of time, while the number of shapes allocated is a similar measure for space. When comparing shapes allocated for versions `BDQ` and `DL`, keep in mind `BDQ` uses much more space as it also allocates quadratic-size tables to lookup bounds, which is not reflected in the shape count.

Time and space for simulated data are plotted in Figure 1. The straight lines of slope roughly 2 in the time plot indicate that over this size range time is growing roughly quadratically in the number of columns. Space grew linearly in the number of columns (and also the number of rows [24]).

7.1.2 *Ceiling phenomenon in the growth of shapes*

The most surprising result from these experiments is that with version `DL` on instances with a fixed number of columns, the observed number of shapes does not grow with the num-

Table 3: Relative error in optimistic and pessimistic alignment-scores compared to the exact score. The *presumed* score is the value obtained by evaluating the optimistic and pessimistic recurrences, while the *actual* score is the real value of the alignment recovered by the heuristic. Statistics are over all splits of an `MVF` benchmark.

				Optimistic error (%)						Pessimistic error (%)					
	Exact score			presumed			actual			presumed			actual		
benchmark	mean	max	dev	mean	max	dev	mean	max	dev	mean	max	dev	mean	max	dev
`glob12`	73,771	83,503	8,979	-0.18	-0.56	0.07	0.13	0.60	0.10	0.01	0.05	0.01	0.00	**0.02**	0.00
`pro12`	74,650	82,968	9,111	-0.90	-1.45	0.11	0.28	0.81	0.15	0.18	0.35	0.03	0.01	**0.16**	0.01
`rh12`	83,518	92,013	10,190	-0.97	-1.37	0.09	0.33	0.90	0.13	0.78	1.02	0.07	0.04	**0.52**	0.04
`kin12`	149,374	165,131	18,222	-0.67	-0.99	0.07	0.21	0.71	0.08	0.36	0.53	0.04	0.02	**0.15**	0.02

Table 4: Relative error in the number of gaps using optimistic and pessimistic versus exact gap-counts. When measuring relative error, we take the absolute value of the difference between the heuristic and exact count. Presumed counts, actual counts, and statistics are as in Table 3.

	Exact			Optimistic error (%)						Pessimistic error (%)					
	gap count			presumed			actual			presumed			actual		
benchmark	mean	max	dev	mean	max	dev	mean	max	dev	mean	max	dev	mean	max	dev
`glob12`	179	227	25	12.31	36.00	4.76	3.45	19.32	2.74	0.48	1.72	0.39	0.10	**1.33**	0.22
`pro12`	369	446	47	25.36	37.87	3.14	4.66	21.51	2.69	3.86	12.38	1.03	0.50	**6.86**	0.62
`rh12`	471	549	61	23.09	31.49	2.37	5.19	31.03	2.60	16.51	31.63	1.75	0.63	**20.04**	1.10
`kin12`	649	753	83	20.92	28.98	2.13	4.38	17.36	2.03	9.72	15.20	1.14	0.43	**5.44**	0.53

ber of rows once a threshold is reached. In our experiments this threshold is small (around 10 rows) and seems independent of the number of columns. This *ceiling phenomenon* in the growth of shapes is shown in Figure 2. Note this contrasts with the exponential growth that occurs in the worst case. We suspect the tractability of Aligning Alignments in practice is largely due to the ceiling phenomenon.

In benchmark alignments, gaps often line up across rows. To test if this causes the ceiling phenomenon, we looked at shape counts in simulated data. Since the generator places gaps randomly and independently in each row, gaps tend to not line up. As shown in Figure 2, the number of shapes hits a ceiling in simulated data as well.

7.2 Accuracy of the optimistic and pessimistic heuristics

Two fast and easy-to-implement alternatives to the exact algorithm are what Kececioglu and Zhang [25] call the *optimistic* and *pessimistic* heuristics, which respectively undercount or overcount the true number of gaps. (The pessimistic heuristic is equivalent to the quasi-natural gap counts of Altschul [1].) Now that we have a provably correct algorithm that feasibly computes optimal alignments, we can measure the accuracy of these heuristics.

Score accuracy The optimistic and pessimistic heuristics give a *presumed* score, the score obtained from their recurrences, and an *actual* score, the score of the alignment recovered by the heuristic.

Table 3 summarizes the relative error between these heuristic scores and the true score from the exact algorithm. As can be seen, the heuristic scores tightly bound the true score (which explains the effectiveness of bound pruning). On average the actual pessimistic score was the most accurate, with less than 1 percent error on all trials. Error seems to correlate with computational difficulty (in terms of time and space of the exact algorithm) as seen in the roughly equal-size benchmarks `glob12`, `pro12`, and `rh12`: as these instances get harder, the heuristics become less accurate.

Gap-count accuracy In contrast to score accuracy, the heuristic gap counts can differ considerably from the true count, as shown in Table 4. The actual pessimistic count was the most accurate on average, yet hit 20 percent error on the toughest benchmark.

8. CONCLUSION

While Aligning Alignments is NP-complete, the exact algorithm with linear-space *dominance pruning* can compute optimal solutions for highly-gapped instances with 100 sequences and 1,000 columns in both alignments, in about 40 minutes and 2 megabytes on a current workstation. The addition of *bound pruning* further reduces the time by an order of magnitude, but increases the space to quadratic.

Many interesting questions remain: the accuracy of the optimistic and pessimistic heuristics compared to the exact algorithm in recovering correct *gap placement*, whether Aligning Alignments is *approximable* within a constant factor (we suspect not), and whether the *ceiling phenomenon* can be explained by analyzing the expected shape-list length under dominance pruning.

9. REFERENCES

[1] Altschul, S. "Gap costs for multiple sequence alignment." *Journal of Theoretical Biology* 138, 297–309, 1989.

[2] Anson, E. and E. Myers. "`ReAligner`: a program for refining DNA sequence multi-alignments." *Journal of Computational Biology* 4:3, 369–383, 1997.

[3] Bafna, V., E. Lawler and P. Pevzner. "Approximation algorithms for multiple sequence alignment." Proceedings of the 5th *Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science 807, 43–53, 1994.

[4] Berger, M. and P. Munson. "A novel randomized iterative strategy for aligning multiple protein sequences." *Computer Applications in the Biosciences* 7, 479–484, 1991.

[5] Carrillo, H. and D. Lipman. "The multiple sequence alignment problem in biology." *SIAM Journal on Applied Mathematics* 48, 1073–1082, 1988.

[6] Feng, D. and R. Doolittle. "Progressive sequence alignment as a prerequisite to correct phylogenetic trees." *Journal of Molecular Evolution* 25, 351–360, 1987.

[7] Fitch, W. and T. Smith. "Optimal sequence alignments." *Proceedings of the National Academy of Science USA* 80, 1382–1386, 1981.

[8] Fredman, M. "Algorithms for computing evolutionary similarity measures with length independent gap penalties." *Bulletin of Mathematical Biology* 46:4, 553–566, 1984.

[9] Garey, M. and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W.H. Freeman and Company, New York, 1979.

[10] Garey, M., D. Johnson and L. Stockmeyer. "Some simplified NP-complete graph problems." *Theoretical Computer Science* 1, 237–267, 1976.

[11] Dayhoff, M.O., R.M. Schwartz, and B.C. Orcutt. "A model of evolutionary change in proteins." *Atlas of Protein Sequence and Structure* 5, 345-352, 1978.

[12] Gotoh, O. "An improved algorithm for matching biological sequences." *Journal of Molecular Biology* 162, 705–708, 1982.

[13] Gotoh, O. "Alignment of three biological sequences with an efficient traceback procedure." *Journal of Theoretical Biology* 121, 327–337, 1986.

[14] Gotoh, O. "Optimal alignment between groups of sequences and its application to multiple sequence alignment." *Computer Applications in the Biosciences* 9:3, 361–370, 1993.

[15] Gotoh, O. "Further improvement in methods of group-to-group sequence alignment with generalized profile operations." *Computer Applications in the Biosciences* 10:4, 379–387, 1994.

[16] Gotoh, O. "Significant improvement in accuracy of multiple protein sequence alignments by iterative refinement as assessed by reference to structural alignments." *Journal of Molecular Biology* 264, 823–838, 1996.

[17] Gupta, S., J. Kececioglu and A. Schäffer. "Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment." *Journal of Computational Biology* 2:3, 459–472, 1995.

[18] Gusfield, D. "Efficient methods for multiple sequence alignment with guaranteed error bounds." *Bulletin of Mathematical Biology* 55:1, 141–154, 1993.

[19] Hirosawa, M., Y. Totoki, M. Hoshida and M. Ishikawa. "Comprehensive study on iterative algorithms of multiple sequence alignment." *Computer Applications in the Biosciences* 11:1, 13–18, 1995.

[20] Hirschberg, D. "A linear space algorithm for computing maximal common subsequences." *Communications of the ACM* 18:6, 341–343, 1975.

[21] Jiang, T., E. Lawler and L. Wang. "Aligning sequences via an evolutionary tree: complexity and approximation." Proceedings of the 26th ACM *Symposium on the Theory of Computing*, 760–769, 1994.

[22] Kececioglu, J. "The maximum trace problem in multiple sequence alignment." Proceedings of the 4th *Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science 684, 106–119, 1993.

[23] Kececioglu, J., H.-P. Lenhof, K. Mehlhorn, P. Mutzel, K. Reinert and M. Vingron. "A polyhedral approach to sequence alignment problems." *Discrete Applied Mathematics* 104, 143–186, 2000.

[24] Kececioglu, J. and D. Starrett. "Aligning alignments exactly." Technical Report 03-12, Department of Computer Science, The University of Arizona, September 2003.
`http://www.cs.arizona.edu/~kece/papers/KS03.pdf`

[25] Kececioglu, J. and W. Zhang. "Aligning alignments." Proceedings of the 9th *Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science 1448, 189–208, 1998.

[26] Lipman, D., S. Altschul and J. Kececioglu. "A tool for multiple sequence alignment." *Proceedings of the National Academy of Science USA* 86, 4412–4415, 1989.

[27] Ma, B., Z. Wang and K. Zhang. "Alignment between two multiple alignments." Proceedings of the 14th *Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science 2676, 254–265, 2003.

[28] Maier, D. "The complexity of some problems on subsequences and supersequences." *Journal of the ACM* 25:2, 322–336, 1978.

[29] McClure, M., T. Vasi and W. Fitch. "Comparative analysis of multiple protein-sequence alignment methods." *Molecular Biology and Evolution* 11, 571–592, 1994.

[30] Myers, E. and M. Jain. "Going against the grain." Proceedings of the 3rd *South American Workshop on String Processing*, International Informatics Series 4, 203–213, 1996.

[31] Myers, E. and W. Miller. "Optimal alignments in linear space." *Computer Applications in the Biosciences* 4, 11–17, 1988.

[32] Myers, E., S. Selznick, Z. Zhang and W. Miller. "Progressive multiple alignment with constraints." *Journal of Computational Biology* 3:4, 563–572, 1996.

[33] Pevzner, P. "Multiple alignment, communication cost, and graph matching." *SIAM Journal on Applied Mathematics* 52, 1763–1779, 1992.

[34] Powell, D., L. Allison and T. Dix. "Fast, optimal alignment of three sequences using linear gap costs." *Journal of Theoretical Biology* 207, 325–336, 2000.

[35] Ravi, R. and J. Kececioglu. "Approximation algorithms for multiple sequence alignment under a fixed evolutionary tree." *Discrete Applied Mathematics* 88, 355–366, 1998.

[36] Reinert, K., J. Stoye and T. Will. "An iterative method for faster sum-of-pairs multiple sequence alignment." *Bioinformatics* 16:9, 808–814, 2000.

[37] Sankoff, D. "Minimal mutation trees of sequences." *SIAM Journal on Applied Mathematics* 78, 35–42, 1975.

[38] Thompson, J., D. Higgins and T. Gibson. "`CLUSTAL W`: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position specific gap penalities and weight matrix choice." *Nucleic Acids Research* 22, 4673–4680, 1994.

[39] Thompson, J., F. Plewniak and O. Poch. "A comprehensive comparison of multiple sequence alignment programs." *Nucleic Acids Research* 27:13, 2682–2690, 1999.

[40] Thompson, J., F. Plewniak and O. Poch. "`BAliBASE`: a benchmark alignment database for the evaluation of multiple alignment programs." *Bioinformatics* 15:1, 87–88, 1999.
`http://www-igbmc.u-strasbg.fr/BioInfo/BAliBASE2/`

[41] Wang, L. and D. Gusfield. "Improved approximation algorithms for tree alignment." Proceedings of the 7th *Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science 1075, 220–233, 1996.

[42] Wang, L. and T. Jiang. "On the complexity of multiple sequence alignment." *Journal of Computational Biology* 1, 337–348, 1994.

[43] Wareham, T. "A simplified proof of the NP- and MAX SNP-hardness of multiple sequence tree alignment." *Journal of Computational Biology* 2:4, 509–514, 1995.

[44] Waterman, M. *Introduction to Computational Biology: Maps, Sequences, and Genomes*. Chapman and Hall, London, 1995.

[45] Waterman, M. and M. Perlwitz. "Line geometries for sequence comparisons." *Bulletin of Mathematical Biology* 46, 567–577, 1984.